HW2: Language Modeling

Srivatsan Srinivasan
srivatsansrinivasan@g.harvard.edu

Sebastien Baur
sebastienbaur@g.harvard.edu

Bernd Huber
bhb@seas.harvard.edu

February 14, 2018

Kaggle Team Name Entries (3 entries) [1] : jiwu jai, Bernd, Sebastien Baur

Github Repo : `https://github.com/harvard-ml-courses/cs287-s18-sb_ss`

1 Introduction

The given problem is a language modeling task which could be loosely defined as predicting the next word that succeeds a sequence of words that we have already seen. This task is a fundamental problem in natural language and has far-reaching applications across part-of-speech tagging, speech recognition, parsing and machine translation. We are performing this task on the Penn Treebank dataset which is a corpus of English sentences and we finally test it on a newspaper article, trying to predict different parts of sentences. We comparatively study n-gram interpolation models, simple Neural network language model, LSTM, GRU and other extensions. We use perplexity as the standard evaluation metric across all these models. We proceed to provide a mathematical description of the problem, model/algorithm description and then elucidate the experiments performed on the models followed by a qualitative discussion of each model variant via the perplexity results calculated on the validation set.

2 Problem Description

The given problem here is a sequence modeling task. The problem could be formalized using the following notations.

- $\mathbf{y}_t \mid t \in \{0, 1, ...\}$ - Vector indicating the word at the t-th position in the sequence. t is the temporal indicator.

[1] Apologies for multiple team entries. We were little less coordinated on who uploads to Kaggle.

- $\boldsymbol{y}_t^e$ - Embedded vector $\boldsymbol{y}_t$ on to different dimension space.
- σ, S, Φ - Sigmoid,softmax and feature transformation functions.
- $\mathcal{V}$ - Vocabulary of the language
- $H(x)$ - Temporal history of words that preceded x.

The final language modeling problem could be interpreted as inferring $P(\boldsymbol{y}_t|\boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2}....\boldsymbol{y}_0)$ through different models. We try to minimize the cross-entropy loss of these predictions while training as it finally generates class probabilities across all words in $\mathcal{V}$.

3 Model and Algorithms

This section will describe the intuition and mathematical formulation behind the models while the following section will elucidate on different variants of these base models and the corresponding results obtained on our dataset.

3.1 Trigram Interpolation Model

Prediction of $\boldsymbol{y}_t$ could just be treated as a unigram, bigram or trigram model. Here, we are using a linearly weighed ensemble prediction among the three models. The MLE estimates of these generative models could be modeled via simple count data. The three models could be formalized as

- **Unigram Model** - Let $C(w)$ be the count of the word entry w. Let N be the total

$$U(\boldsymbol{y}_t) = p(\boldsymbol{y}_t = w) = \frac{C(w)}{\sum_{i=1}^{\mathcal{V}} C(w)}$$

- **Bigram Model** - Let $C(w)$ be the count of the word entry w. Let N be the total

$$B(\boldsymbol{y}_t) = p(\boldsymbol{y}_t = w|\boldsymbol{y}_{t-1}) = \frac{C(w|\boldsymbol{y}_{t-1})}{\sum_{i=1}^{\mathcal{V}} C(w|\boldsymbol{y}_{t-1})}$$

- **Trigram Model** - Let $C(w)$ be the count of the word entry w. Let N be the total

$$T(\boldsymbol{y}_t) = p(\boldsymbol{y}_t = w|\boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2}) = \frac{C(w|\boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2})}{\sum_{i=1}^{\mathcal{V}} C(w|\boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2})}$$

The intuition behind this idea is that language is essentially connected and the next word in the sequence could be generated independently, or generated based on the previous word(eg : blue sky) or generated based on the previous two words(for instance). We then take a convex combination of these three probabilities to predict our next word. Let history H(w) represent the words that preceded the word w.

$$P(\boldsymbol{y}_t|H(\boldsymbol{y}_t)) = \alpha_1 * P(\boldsymbol{y}_t|\boldsymbol{y}_{t-1}, \boldsymbol{y}_{t-2}) + \alpha_2 * P(\boldsymbol{y}_t|\boldsymbol{y}_{t-1}) + (1 - \alpha_1 - \alpha_2) * P(\boldsymbol{y}_t) \mid \alpha_1, \alpha_2 \in [0, 1]$$

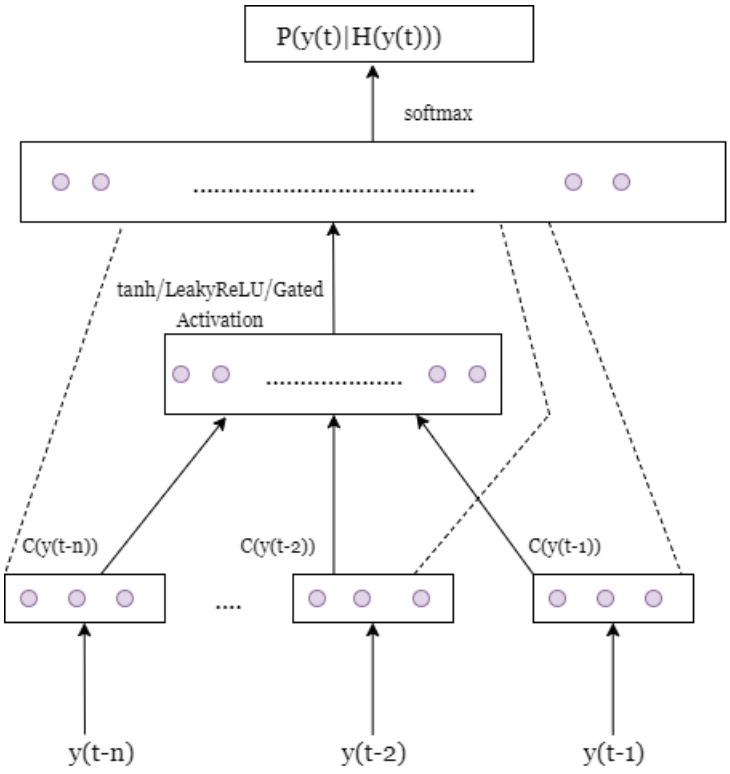

Figure 1: Model Architecture of standard NNLM. Dotted lines represent skip through connections.

3.2 Neural Network Language Model

In a Neural Network Language Model (NNLM, Bengio et al. (2003)), we try to model the aforementioned conditional probability P($\boldsymbol{y}_t|H(\boldsymbol{y}_t)$) using a feed-forward neural network. We choose a fixed look-back window/history of n words and use this history as input feature vectors. More precisely,

$$P(\boldsymbol{y}_t|H(\boldsymbol{y}_t)) = \Phi(\boldsymbol{y}_{t-1}....\boldsymbol{y}_{t-n})$$

The feature transformation Φ could be decomposed into two functions.

- Embedding - A function $C(\boldsymbol{y}_i)$ which maps the one-hot word vector to an embedding of fixed dimension $\mathcal{D}$
- Another function f which outputs the probability distribution over $\mathcal{V}$ words for $\boldsymbol{y}_t$.

Both of these functions could be combined into a single functional form for Φ

$$\Phi(\boldsymbol{y}_{t-1}, ...\boldsymbol{y}_{t-n}) = f(C(\boldsymbol{y}_{t-1}), C(\boldsymbol{y}_{t-2})...C(\boldsymbol{y}_{t-n}))$$

The loss function for this neural network is its negative log likelihood(NLL) and we also incorporate a L2 weight decay penalty in our model for better regularization. Besides, we have tried both variants of embeddings - static and dynamic(trained along with the network). The hyperparameters and different experiments that we constructed with this model have been elucidated in the following section.

This neural network architecture is expected to perform better than the trigram model because it is able to capture non-linearities thanks to the feedforward network. On the other hand, this neural network might suffer from scale when we need to look back at a longer window of text. For instance, a sentence could proceed as follows. "I have been residing in France for the last 25 years, finishing my high school and college in this country and hence, I am highly fluent in speaking ——". The next word in the sentence is highly likely to be "French" and the connection could be inferred from words occurring much earlier in the sentence, which a short window NNLM might not be able to capture. Increasing the window size would mean more parameters and would make it difficult to train for the model. Hence, we need some representation which retains the essential information that occurred right from the beginning of a sentence, but in a compact continuous vector space, so that it could be used along with the remaining feature vectors to predict the next word in the text. This approach forms the backbone of recurrent neural networks.

3.3 RNN/LSTM/GRU

3.3.1 Simple RNN

With a recurrent neural network, we would like to encode the essential information of the sentence through the hidden layers. The hidden layer at the previous temporal step is coupled with the current time step's feature vector to train the current hidden layer and make predictions simultaneously. Let us denote the output of the neural network at any time be $\boldsymbol{o}(t)$. Let h_t^l denote the weight of hidden layer l at time t. In total, let there be L layers in the model. Here h^0 could be thought of as input y_t. In a classical RNN, the transition of the network could be represented through some non-linear activation function f by

$$h_t^l = f(W^{hh} * h_t^{l-1} + W^{ht} * h_{t-1}^l), f \in \{\sigma, \tanh ....\}$$

Here, we encode all the temporal history that was learned by the node through h_{t-1}^l and pass in new information learned from the previous layer in conjunction with it. The expectation is that the hidden state in the previous timestep captures all necessary information of the sentence in its vector space and we can directly learn from that compact representation.

This works fine for smaller networks which deal with very short sequence lengths. When the sequences get longer and we chain together the hidden layers temporally, gradient at time-step t is influenced by the gradients of the temporal predecessors. This presents two problems. If all the neurons in the gradient chain have high absolute values, then we might encounter gradient explosion. On the other side, there is a vanishing gradient problem too where a certain layer h_t in the chain could be switched off and there is no backpropagation to prior temporal layers h_{t-i} from thereon.

3.3.2 GRU

LSTM and GRU were designed to address precisely this problem. GRU(Gated Recurrent Unit) bypasses the vanishing gradient issue by ensuring direct-pass through gates(skip-connections) across the temporally connected hidden units, thus setting up a path of gradient flow even if the standard recurrent chain could run into zero gradients. Thus, we could pick a convex combination of the standard RNN and the skip connection information. This convex combination parameter

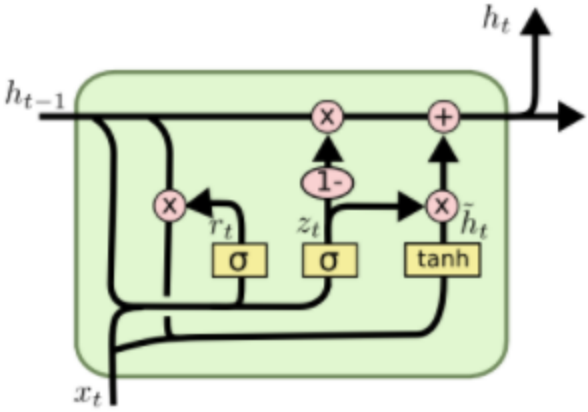

Figure 2: Schematic of GRU. Source: Colah's blog

could be thought of as a gate to the information flow across hidden layers. Besides, in certain cases we need to be able to forget the information coming through after a point in time. Hence, we add a forget gate to the model too that helps it forget the information learned by the RNN hidden layers. Figure 2 elucidates the workings of a GRU through a diagram with certain notational differences from our equations($\alpha = z_t, \beta = r_t$).

All of this could be summarized into the following model.

$$h_t^l = \vec{\alpha} \cdot f_1(W^{ht} h_{t-1}^l \cdot \vec{\beta} + W^{hh} h_t^{l-1}) + (I - \vec{\alpha}) \cdot h_{t-1}^l$$

The vectors $\vec{\alpha}, \vec{beta}$ are the two gates in the GRU, with the former being the and the latter being the reset gate. These gates are probabilities in [0,1] and are themselves trained through feedforward nets with h_{t-1}^l, y_t as features.

$$\vec{\alpha} = f_2(W^{ah} * h_{t-1}^l + W^{ai} * y_t)$$
$$\vec{\beta} = f_2(W^{bh} * h_{t-1}^l + W^{bi} * y_t)$$

3.3.3 LSTM

LSTM, on the other hand,maintains short and long term memory through cell states, which are modified with new information added or forgotten based on gates. There are three common gates in the basic variant of LSTM as seen in Figure 3 - a forget gate f_t which suggests what prior context we need to throw out of memory, an input gate i_t which decides what inputs to pass through to update cell state and a final output gate which decides what parts of the cells state that we are going to output. This could be illustrated with the following equations assuming there is only one

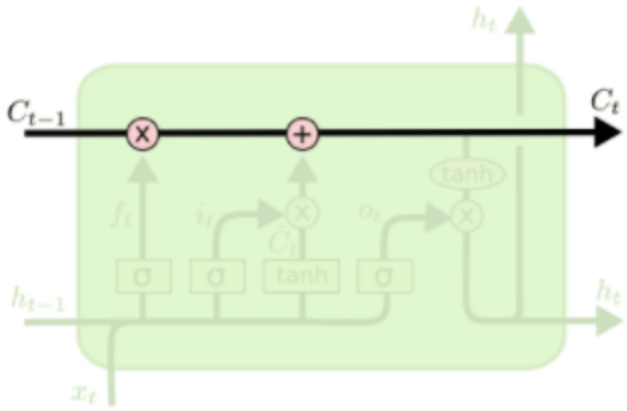

Figure 3: Schematic of LTSM. Source: Colah's blog

hidden layer.

$$
\begin{aligned}
o_t &= \sigma(W_o \cdot [h_{t-1}, y_t] + b) \\
h_t &= \tanh(c_t * o_t) \\
c_t &= f_t * c_{t-1} + i_t * \hat{c}_t \\
\hat{c}_t &= \tanh(W_c[h_{t-1}, y_t] + b) \\
f_t &= \sigma(W_f \cdot [h_{t-1}, y_t] + b) \\
i_t &= \sigma(W_f \cdot [h_{t-1}, y_t] + b)
\end{aligned}
$$

4 Experiments

The performance of our models are described in terms of Kaggle Map score in Table 1 and in terms of validation set perplexity in Table 2. What follows is a qualitative discussion of the experiments and hyperparameters.

Model	**KAGGLE MAP**	**Team Name**
TRIGRAM MODEL	0.30435	jiwu jai
LSTM	0.325	Bernd
NNLM	0.314	Sebastien Baur

Table 1: Table with Kaggle MAP results.

4.1 Notes on Trigram Model

At first we sampled our model using equal weights for the uni-, bi- and trigrams. This approach, however, assumes that in English language, the unigram probabilities are equally important to predict a word than the context. This motivated us to favor the assumption of putting more

Model	Perplexity
TRIGRAM MODEL	225
NNLM	134
LSTM	96
GRU	124

Table 2: Table with Validation Set Perplexity results.

weights on the bi- and trigram models will improve the score.

We optimized the weights using nelder-mead to maximize the log likelihood computed on the validation test. This led to an optimal $\alpha_1 = 0.559375$ and $\alpha_2 = 0.39479167$, leading to a slight improvement of the MAP score on Kaggle to 0.30435 as compared to 0.27 we got with equally weighted combination.

4.2 Notes on NNLM experiments

We conducted several experiments on the NNLM architecture outlined in the previous section, whose results are detailed in Table **??** on Page 8. In the process, we discovered several interesting insights on the impact of activation functions and regularization in our models. As mentioned by Chiu and Nichols (2015), the size of the dimensionality of the embedding doesn't really have a significant impact on the performance of their model as our performance remained largely similar across GloVe 6B 50d embedding and 300d embedding. With the number of parameters in the neural network being comparable to the size of the dataset, we introduced different regularization techniques. As our network was allowed to tune the embedding vectors (it adds many parameters to the network), we use a L_2 penalty, which to reduce overfitting and the results from the L2 penalty were not significant to our model narrative and hence we don't discuss them here.

We also used **dropout** (Srivastava et al. (2014)), and, unexpectedly, it deteriorated the performance of the network as can be seen from Table **??**. It is similar to the pattern we observed in the last assignment too where a dropout of 0.5 seemed to be too high and limits the model from training well, even though it is the standard value used in most papers. We beleive that a smaller dropout value helps our cause better.We also looked at the impact of **batch normalization** (Ioffe and Szegedy (2015)) on the performance of the model. The advantages of batch normalization in training NN is well-studied. It scales the activations in the [-2,2] range, which allows for easier backpropagation of gradients and reduces the importance of the learning rate and weight initialization. This generally translates to higher training performance (both in terms of speed of convergence and final perplexity). With the activations being dependent on the batch(that are randomized), the predictions for a given element are inherently noisy and this randomness allows the model to generalize better, inline with our observations.

As our model wasn't overfitting much, we tried to study the effects of increased dimensions of our network. The performance did not improve drastically particularly in the absence of both batch norm and dropouts (154→149, 166→161, ...). When you add dropout or batch normalization, the model is able to take full advantage of the increased capacity and we see larger improvements

hdim	context size	activation	dropout	train (BN)	train (no BN)	val (BN)	val (no BN)
50	5	gated	0	103	99	144	154
100	5	gated	0	84	97	136	149
50	5	lrelu	0	105	104	151	166
100	5	lrelu	0	91	118	142	161
50	5	gated	0.5	232	224	191	182
100	5	gated	0.5	210	202	174	166
50	5	lrelu	0.5	260	248	209	203
100	5	lrelu	0.5	224	222	187	186
50	8	gated	0	91	103	140	152
100	8	gated	0	85	92	134	148
50	8	lrelu	0	102	109	144	165
100	8	lrelu	0	92	115	139	159
50	8	gated	0.5	266	221	211	178
100	8	gated	0.5	211	191	171	159
50	8	lrelu	0.5	240	255	198	205
100	8	lrelu	0.5	228	226	186	187

Table 3: Results of different setup of the hyperparameters for the NNLM model. The last four columns indicate the perplexity on train and validation sets with and without batch normalization for the given hyperparameters.

(144→136, 151→142 for similar configurations). It is also worthwhile to note that these improvements are already on top of a well-trained model and it is difficult to squeeze more performance from a well-trained model.

As figuring out what the next word requires the model to take into account long term dependencies, we tried to increase the context size, comparing the results between 5 and 8. Generally, this higher context size boosts the performance of the model. As the input of the model potentially contains an end token in the middle of the context (with all the words before the token being irrelevant for the current prediction), we wanted the model to be able to easily zero out everything before the end token. This is why we experimented with two non-linearities: a.) the classical leaky-relu as compared to b.) gated linear units (recently preferred by Gehring et al. (2017) over the classical gates in RNN). The intuition behind that choice was that it was probably easier to zero out useless parts of the input (as the words before the end token) using a multiplicative nonlinearity than by a more classical one. It turns out that this nonlinearity proved a significant value-add to our performance. We attribute this better performance to the more complex interactions that can be captured by a multiplicative interactions, and to the higher number of hyperparameters that we used.

4.3 Notes on RNN experiments

In our experiments, we did not try simple RNNs because we envisioned that we would run into vanishing gradients issue. Hence, we directly began experimenting with LSTM largely with a little quick trial with GRU. There were several important facets of implementing a deep LSTM that

n_layers	Hidden dim	Dropout Embedding	Validation PPL
1	300	yes	109
2	300	yes	109
2	300	no	96
1	100	no	119
1	300	no	102
2	100	no	125

Table 4: Results from LSTM. Performance doesn't change drastically when capacity is increased.

we learned in our implementation.

In the first iteration, we faced a memory overflow and we later detected that one large culprit was the perseverance of the computation graph in memory. Hence, we detached the hidden nodes after each batch and then re-initialized them with these values recursively in each new batch. Then, we faced an exploding gradient issue particularly on increasing the dimensions above 100 which we fixed by clipping the gradients. We then saw that the model over-fitted and we added dropout and L2 norm penalty on weights. With all these features in place, we ran a few structured experiments with the hyperparameters(not in the same scale as NNLM as these LSTMs took much longer 1hr to train). We still faced memory issues when loading deeper networks(more than one layer) which we finally addressed by turning cudNN off.

Largely, we were analyzing the impact of the hidden layer dimension, the impact of dropouts in different layers and the depth of the layers. Lot of our hyperparameters are inspired from the work of Zaremba et al. (2015). In all our experiments, there was dropout after the output before projecting to the vocabulary space, the word embedding was dynamically trained(which added non-trivial value to our performance compared to static embeddings)and the L_2 penalty coefficient was 0.001. We also consistently clipped the gradients above a norm of 5(batch normalized).

As mentioned in the previous subsection, too much dropout adversely affects the model. Adding dropout just after embedding prevented our model to improve even when additional layers were added. The best PPL score of96 could be achieved when we removed the dropouts after embedding. We also observe that increasing the capacity of the model in terms of bigger hidden layer or deeper layers does not increase the performance of the model drastically. These aspects could be visualized in Table 4. Besides, we could not perform too many experiments on GRU owing to lack of time and we could not tune its hyperparameters with the same amount of rigor that we did for LSTM. The results in the table 2 are for a 1 layered, 150 dimensional GRU with similar other regularization parameters and with little hyperparameter tuning, it was able to achieve comparable performance with the best NNLM model.

Few works of research tackling these problems seem to be promising, but we didn't get the chance to implement them. In particular, batch normalization for hidden connections Cooijmans et al. (2016), as well as skipped connections He et al. (2016), Abolafia et al. (2018), might help the gradients to bflow through the network. Interestinngly, the significant improvement in perplexity did not translate into significant improvement in Kaggle MAP score. We believe it is so because the PPL score does not have any implicit ranking based score and is smooth across all solutions in vocabulary while the MAP score was rigid.

5 Conclusion

In this exercise we developed sequence models using trigram interpolation methods, neural network language models and recurrent neural networks(GRU/LSTM). The trigram interpolation was our baseline estimate and performed surprisingly well with the right hyperparameter tuning for all its perceived theoretical limitations. The perplexity results of the Neural Network Language Model(NNLM) with gated activation units and batch normalization improved the perplexity score significantly compared to previous model as it was able to capture the text information over a bigger context window and use adaptive non-linear units to infer the conditional probabilities. The recurrent units outperformed the NNLM, thanks to their long term memory retention properties. wHile the GRU units performed marginally better than the NNLM since we did not spend significant time in tuning this model, the **LSTM** model with its forget, input and output gated RNNs, far outperformed the NNLM clearly with the **best PPL score of 96**(matching the performance of Zaremba et al. (2015) while the minor discrepancy could be attributed to our smaller network size) and emerged as a clear winner in the problem of sequence modeling. Through this exercise, we clearly understood the impact of hyperparameters on really deep networks as we could almost halve the perplexity score by fine-tuning the hyperparameters and regularizing the networks appropriately.

References

Abolafia, D. A., Norouzi, M., and Le, Q. V. (2018). Neural program synthesis with priority queue training. *CoRR*, abs/1801.03526.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, pages 1137–1155.

Chiu, J. P. C. and Nichols, E. (2015). Named entity recognition with bidirectional lstm-cnns. cite arxiv:1511.08308.

Cooijmans, T., Ballas, N., Laurent, C., and Courville, A. C. (2016). Recurrent batch normalization. *CoRR*, abs/1603.09025.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. (2017). Convolutional sequence to sequence learning. In *ICML*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

Zaremba, W., Sutskever, I., and Vinyals, O. (2015). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329v5*.